CSDL-C-5694

AIPS Language Trade Study

by

Alton A. Knosp, Jr.

March 1984

Approved: P. Felleman

The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts 02139

ACKNOWLEDGEMENT

This report was prepared by The Charles Stark Draper Laboratory, Inc. under Contract NAS9-16023, Task Order 35, with the Lyndon B. Johnson Space Center of the National Aeronautics and Space Administration.

Personnel who contributed extensively to the research for and review of this document include Roger Racine, Dr. J. Barton De Wolf, James Kernan, Glen Ogletree, and Mary Whalen.

ABSTRACT

This report presents a comparison of the advantages and disadvantages of choosing each of the programming languages Ada[1], C, FORTRAN 77, HAL/S, JOVIAL J73, and Pascal as the selected language for the proof-of-concept demonstration of the NASA/JSC Advanced Information Processing System (AIPS). Each language, in its turn, was examined for its potential impact on the economical development and maintenance of software that would meet the goals and characteristics emphasized by the system requirements.

The specific criteria for evaluating each language were chosen based on:

- "AIPS System Requirements" (October 12, 1983) CSDL Report number AIPS-83-50
- Modern studies of software reliability
- Experiences with large software programs at CSDL.

The data for this language trade study were collected during the AIPS Technology Survey which was conducted for the NASA/JSC by CSDL as part of the AIPS Program Phase I activities. The language data were drawn from conferences, personal experiences, private meetings, and telephone conversations as well as text books, language standards, and technical journals as listed in the bibliography.

Comparisons between the six languages indicated that Ada was the best choice for designing and coding the AIPS proof-of-concept demonstration.

---

1 Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

TABLE OF CONTENTS

1.0 INTRODUCTION AND SUMMARY

The purpose of this study was to select the language to be used to implement the proof-of-concept demonstration for the Advanced Information Processing System (AIPS) under development by the Charles Stark Draper Laboratory, Inc. (CSDL) for the Johnson Space Center of the National Aeronautics and Space Administration (NASA/JSC). Six languages were evaluated for features that would support the characteristics of the AIPS architecture that were emphasized by the system requirements [1]. After comparing these evaluations, Ada was the language selected.

The languages used for comparison in this study were Ada, C, FORTRAN 77, HAL/S, JOVIAL J73, and Pascal. These languages were chosen based on their past, current, and/or projected use for embedded avionics software and microprocessor applications. Other languages were excluded due to either less common usage, similarities to a language already chosen, or specialization for a narrower range of applications than required by the AIPS program.

The criteria for evaluating each language covered five basic areas:

- Error detection, error handling, and error containment
- Modularity and separate compilation
- The ability to provide real-time system constructs
- The stability and portability of the language
- The availability of software development tools.

The criteria were chosen for their impact on the characteristics emphasized in the system requirements [1] for the AIPS architecture as they apply to software, specifically, economic development and maintenance of reliable, testable, manageable software that is flexible with respect to change and growth.

Comparisons were not made between specific implementations of languages. Rather, the standards for the languages were compared based on the evaluation criteria. Implementations were considered as available development tools and as evidence of language stability and portability.

This document is organized in five sections as follows: Section 2 gives a brief description of the AIPS program and requirements, Section 3 describes the criteria chosen for the evaluations and comparison of the languages. Section 4 presents the evaluations. Section 5 presents the conclusions drawn from comparing the evaluations.

The principal conclusion reached by this study was that Ada is the best language choice for the AIPS proof-of-concept design and implementation. This resulted from comparing the evaluations of each language in terms of each of the five criteria cited above.

2.0 AIPS PROGRAM DESCRIPTION

This program will develop and demonstrate a system architecture and the associated design and evaluation methodologies to serve the need for advanced information processing across a broad spectrum of future NASA missions. The architecture will incorporate relevant advanced hardware and software technologies and will provide integrated operation, reliability and fault tolerance, testability, manageability, and flexibility for growth and change. The last of these is of particular significance because the architecture must be adaptable to a range of applications for both space and aircraft missions. Also, mission needs and requirements are expected to change in time, and the system design developed by the AIPS effort is expected to provide the flexibility to accommodate such changes with minimum redesign, reverification and revalidation.

The emphasis of the AIPS Program will be a proof-of-concept demonstration of a system architecture which will be implemented with available component-level technologies. The design methodology, hardware/software architecture, system modularity, and validation processes are significant elements of the output. The attribute of change tolerance is expected to permit the insertion of new technologies as well as permit graceful modification to support various specific applications as mentioned above.

The AIPS program should not be characterized as specific to any particular NASA application or new initiative, but rather as a system-level pathfinder effort that is independent of other programs. It is expected that major technology exchange between this research and development program and specific NASA application programs will be a significant benefit.

2.1 AIPS PROGRAM TECHNICAL APPROACH

The AIPS architecture, design methodology and testing techniques must provide, in addition to traditional figures of merit, adaptability to the inevitable changes and growth in mission-specific requirements. The AIPS concept is intended to permit mission avionics to be assembled from a set of previously validated hardware and software components. This requires component and system ability to detect and recover from software errors, to mask software data errors, and to use modern techniques for verification and validation, such as the enforcement of high-order language features. Changes in mission requirements, then, will impact only a limited subset of hardware and software elements. The changes become additions or deletions to the avionics system which do not result in reverification or revalidation of the complete hardware and software system.

In order to achieve the total set of desired system attributes, the AIPS proof-of-concept system will be configured as a distributed system. The intent is to provide considerable total-system capability through the delegation of functions to multiple processing sites, each of which requires only modest capability. Functional partitioning within the mul-

tiple processing sites will be determined by: the source, destination, and character of data transfers; the physical location of system elements; and the functional relationships between elements. Adherence to available, broadly accepted hardware and software standards (e.g., instruction set, communication protocol and high-order language) will be emphasized, as will extensive utilization of advanced design and evaluation methodologies. The demonstration will be constrained to use only one source language and one instruction set architecture.

3.0 CRITERIA FOR EVALUATION AND COMPARISON OF LANGUAGES

The criteria used to compare languages were chosen for their impact on the goals emphasized by the AIPS System Requirements, namely, reliability, manageability, testability, cost, integrated operation, fault tolerance, and flexibility. We divided these criteria into five basic areas:

1. Error detection, error handling, and error containment
2. Modularity and separate compilation
3. Provision for real-time system programming constructs
4. Stability and portability
5. The availability of software development tools.

These are described in more detail below.

3.1 ERROR DETECTION, ERROR HANDLING, AND ERROR CONTAINMENT

The ability to detect errors, respond to errors during execution, and to contain the effects of errors will directly impact both the cost and the reliability of AIPS software over its entire life-cycle. The cost of correcting an error will tend to increase exponentially from the time the error occurs until it is detected and corrected; this is due to increasing interdependence of related source code, documentation, and tests over time, and the need to modify these items to implement the correction. Limiting the scope of influence of an error could diminish the acceleration of this cost. Eliminating and avoiding errors that tend to propagate boundlessly could further reduce cost and increase reliability. Further, for reliability, AIPS software will need to gracefully respond to potential errors that cannot be eliminated.

In the detailed discussions which follow, we consider four criteria which affect the detection, reduction, and handling of errors:

1. Design translation
2. Static error detection
3. Dynamic error detection and exception handling
4. Error containment

3.1.1 Design Translation

The complexities of translating a design into a source code language will affect the number of errors that occur during this task. Basic language primitives that correspond to the design abstractions used by most applications, such as data types and operations, could simplify this task. Too many primitives, however, increase the probability of translation errors due to the complexity of the source code language. No matter how many primitives are provided by a language, some flexibility for defining data types and operations would be required to allow for any unforeseen design abstraction. The ability to define constructs not provided by a language and to encapsulate these definitions so that they may be used as primitives would balance the tradeoffs between too many and too few primitives for design abstractions.

To evaluate the expressiveness of each language, we examined what primitives could be used directly as applications design abstractions and what primitives could be used to construct and encapsulate abstract data types and operations. These included built-in data types, operations for built-in data types, constructs for defining user-defined data types, built-in operations for user-defined data types, and primitives for constructing and encapsulating user-defined operations. Throughout this paper we use the following generic terms to refer to classes of the more common built-in application data types:

integer	whole numbers
float	floating point, real, and fixed point data types
boolean	single bit or flag types with exactly two possible values
character	representations of single characters of text
string	built-in array types with bit or character components generally indexed by integer values.

Other built-in types useful for applications programming are explicitly stated. Generic terms used to refer to classes of built-in operations for built-in data types include:

basic arithmetic	addition, subtraction, multiplication, division, and additive inverse
exponential	exponent and log functions or operations
trigonometric	sine, cosine, tangent, and the like
hyperbolic	hyperbolic trigonometric functions
relational	greater-than, less-than, equal-to, not-equal-to, greater-than-or-equal-to, and less-than-or-equal-to
miscellaneous arithmetic	absolute value, sign, and others

logical	the Boolean operations AND, OR, equivalence, and logical inverse

The following terms are used to designate primitives commonly used for defining user-defined types:

precision specification	used to define a subset of the set of possible values for float data types by a precision rule
array	a single- or multi-dimensional aggregate where each component is the same data type and the components can be indexed by a finite range of discrete values
Cartesian product	an aggregate, known as a structure or record in many languages, with components of one or more different data types
address	a data type for constructing linked lists and for accessing dynamically allocated data objects
enumerated list	a data type consisting of a finite set of ordered, user-defined values which may be used in relational operations and assignment
range	an ordered subset of the set of possible values for a data type.

Primitives for encapsulating the definitions of user-defined data types and/or operations included, but were not limited to:

procedure	a named module, used to encapsulate an algorithm, that can be invoked by name
function	a module, used to encapsulate an operation, that is invoked by name, may accept a number of inputs, and returns a value.
task	a module that encapsulates an algorithm which may logically execute simultaneously with other modules, i.e., as if it were executing on a separate processor
data block	a module used to encapsulate data or data types
block	a module, used to encapsulate an algorithm, that can be treated as one statement and/or used to bound the range of scope for another construct

To evaluate the simplicity and understandability of each language, we examined limitations for specifying identifier names, syntax for comments, the number of key words, and the number of key words that could be understood as English. Also examined were the syntax of control struc-

tures, the use of punctuation, and the use of cryptic or non-English key words.

3.1.2 Static Error Detection

Detection and correction of errors at compile time would reduce the effort spent in testing. Eliminating these errors would decrease the cost of configuration management through the reduction of the number of different object modules, load modules, output listings, and data produced. Testing and reliability would also be improved due to the minimization of the number of errors and the kinds of errors that need to be detected through testing.

Each language was examined for strong type checking and explicit type specification. Strong type checking disallows implicit conversions, even for different types with the same implementation. This forces a conversion to be coded explicitly for an expression to be assigned to a variable or parameter of a different data type. Typographical errors, unintentional implicit conversions, and errors due to identical variable names in different scopes, for example, would otherwise be difficult to detect and locate even at run-time.

Explicit type specification allows the programmer to specify precisely what values a data type may assume. The use of range limited integers and enumerated list types to define array bounds and iterative loop control would allow a compiler to check for out-of-bounds conditions, even to the point of eliminating run-time checks for such conditions in loops and blocks of code. By restricting address types, a similar savings would be possible for addressing operations and linked lists.

3.1.3 Dynamic Error Detection and Exception Handling

There are classes of errors that are difficult or impossible to detect statically. Developing code to dynamically detect and correct these errors would significantly increase the cost of the AIPS software development effort. Constructs that automatically detect and/or correct such errors would not only reduce the costs but also increase software reliability by reducing or eliminating errors that might otherwise result from the designing, writing, and testing of such constructs.

Each language was evaluated with respect to the kinds of run-time errors that the language would automatically detect. Examples include conversion errors, data object over-flows and under-flows, and references to unallocated data objects. Also evaluated were the means the language provided for handling errors and exceptions. These included branching to blocks of user-defined code within a module or at the end of a module, aborting execution, default correction schemes, and undefined responses. Interrupt detection is considered under real-time system programming constructs and is described in a following section.

3.1.4 Error Containment

Some control and data structures have a potential for causing errors which are difficult to detect and virtually impossible to trace. Yet, it can be argued that there may be a need to use these dangerous constructs. A language could reduce this need by providing alternative structures that are more verifiable and by restricting the use and/or scope of the less verifiable structures.

Each language was evaluated for restrictions on GOTO statements and conditional iteration statements that could cause infinite loops. This included examining the structures used to control iteration and the restrictions and alternatives to the GOTO statement.

Each language was evaluated for side effects. A side effect is the modification of a data object that is not obvious from the use of an operation or module. This could cause errors that are difficult or impossible to detect. Constructs that can cause side effects include global identifiers, call-by-reference parameter passing, and aliasing of variable names. These allow operations and modules to bypass language-defined interfaces to access data and program objects. The use of global identifiers was evaluated concerning where and when an identifier could be accessed (the scope of the identifier), particularly the use of the nesting of modules to restrict this scope. The control and techniques for parameter passing were also evaluated. Call-by-reference parameter passing by itself allows unintentional modifications of an argument while the invoked module is executing. When combined with aliasing, referencing the same data object via different names, errors can occur that are impossible to detect. Call-by-value and call-by-value-result parameter passing avoid such errors. However, implementations for these techniques may not be efficient, especially for user-defined data aggregates. Restrictions on other potential side effects that were built-in to a language were also evaluated. These included address data types and discriminated union data types. (A discriminated union is a Cartesian product data type with a definition that varies according to the value of a designated component.)

3.2 MODULARITY AND SEPARATE COMPILATION

Large programs, such as AIPS, would be virtually unmanageable without the capability of decomposing programs into modules. By encapsulating or "hiding" the implementation of an algorithm or data structure, a module would provide a boundary or interface behind which an implementation could be referenced as a single entity and, with separate compilation, modified and tested without affecting source code in the remainder of the program. Modules also provide boundaries for error containment, testing, coding, documentation, and the division of labor. The manner in which modularity is provided by a language would affect testing, reliability, flexibility, and manageability of the AIPS software development effort.

The following areas were specifically evaluated:

1. The types of modules provided

2. The interfaces between modules

3. The separate compilation and management of module interfaces.

3.2.1 Types of Modules Provided

Each language was evaluated for the kinds of modules that were provided, such as procedures, functions, tasks, data blocks, and statement blocks (See section 3.1.1). The different modules were evaluated for their independence and the ability to be nested within other modules.

3.2.2 Interfaces Between Modules

Each language was evaluated for how modules appeared to the rest of the program and how these interfaces were checked by the language. Particular attention was payed to this criterion since interfaces between modules are one of the largest sources of errors in large programming efforts.

3.2.3 Separate Compilation and Interface Management

Each language was evaluated concerning what modules, if any, could be compiled separately, whether interfaces could be checked between such modules, and how, or if, the language provided any configuration management for these interfaces.

3.3 PROVISION FOR REAL-TIME SYSTEM PROGRAMMING CONSTRUCTS

Since the purpose of AIPS is to provide a basic architecture for the design and development of real-time avionics systems, the ability of a language to provide real-time system constructs directly impacts the amount of effort necessary to design, code, and test the system. The existence of system constructs within a language could eliminate the need to re-create these constructs for each application. General built-in systems constructs could also be used across a wide number of applications.

Each language was examined for existence of the following systems programming constructs:

1. Real-time tasking constructs

2. Constructs to manipulate bits and bytes

3. Constructs for handling primitive input/output (I/O) operations

4. Interrupt handling constructs.

The ability to create new constructs was also evaluated.

3.3.1 Real-Time Tasking Constructs

Each language was evaluated for the existence of real-time tasking constructs. These included modules which are able to execute concurrently, in either a logical or physical sense; constructs to begin execution, suspend execution, resume execution, or cancel execution of these modules; constructs to execute these actions based on time or events; and constructs for inter-module communication and protection against the simultaneous access of a data object by more than one module while the data object is being modified.

Where these constructs existed, they were evaluated for their usefulness and completeness for real-time systems. Where they were lacking, the language was evaluated for the ability to create these constructs with the built-in language constructs.

3.3.2 Ability to Manipulate Bits and Bytes

Each language was evaluated for the ability to manipulate bits and bytes, particularly for the purpose of setting flags in special purpose registers and for low-level I/O functions. The operations considered included setting and resetting specific bits within a byte or word, defining a data type to correspond to a specific bit or byte within a specific area of core, and shifting bits within a byte or word.

3.3.3 Input/Output Capabilities

The capability of performing low-level I/O would be, of necessity, an implementation dependent feature on any embedded computer system. However, the existence of constructs within a language to handle low-level I/O defines, at the least, a common interface through which otherwise general programs could communicate with this implementation-dependent function. Such constructs also eliminate the proliferation of user-defined interfaces that might occur between programs and within the same program.

3.3.4 Exception Handling Capabilities

Each language was evaluated for the existence and usefulness of constructs for handling both hardware exceptions and software exceptions. As with low-level I/O, such constructs enforce a degree of consistency through a program and between programs concerning exception handling.

3.4 STABILITY AND PORTABILITY

The AIPS requirements for flexibility and economy imply first, that the language used for AIPS be stable, and second, that the software be portable to different target machines. The more a language changes the more implementations of the language will differ, even for the same target machine. Modifications and reinterpretations of a language increase the probability that software written for one implementation will require extensive changes in code, documentation, and testing to be validated on another implementation.

The following are areas over which each language was evaluated for its effect on this criterion:

1. The stability of the language standard
2. The enforcement of the language standard
3. The demand for the language.

3.4.1 Stability of the Language Standard

Each language was evaluated in terms of its standard, or what could be considered a standard, for the language and the tendency for the standard to change.

3.4.2 Enforcement of the Language Standard

Each language was evaluated for enforcement of the language standard by determining what body, if any, is responsible for the enforcement, the means it could use to enforce the standard, and what incentive it had to enforce the standard.

3.4.3 Demand for the Language

Finally, each language was evaluated with respect to the demand for the language. The basis for evaluation included, in general terms, how wide-spread was the use of the language, the efforts of companies to provide implementations and tools for the language, and the efforts of other companies to acquire these tools and implementations.

3.5 AVAILABILITY OF SOFTWARE DEVELOPMENT TOOLS

A language will be practicable for AIPS only if a useable implementation exists with relevant software development tools. The costs of implementing and/or designing language compilers and tools could have adverse impact on the economy of a development effort.

We evaluated each language for the availability of three classes of software tools:

1. The availability of implementations targeted to one of a number of candidate processors for AIPS

2. The availability of other implementations, debugging tools, and analysis tools

3. The availability of configuration management tools for managing a large programming effort.

3.5.1 Availability of Implementations for AIPS Processors

The processors that were considered candidates for the AIPS program included:

1. Any possibly available MIL-STD-1750A processor

2. The Intel 8086

3. The Intel IAPx series

4. The Motorola MC68000 series

5. The National 16016 and 32032 series

6. The Zilog Z8000 series.

3.5.2 Availability of General Tools

We compared what compilers, debuggers, and other software support tools were available on mainframe computers and systems capable of storing large amounts of text and programs, such as the VMS and UNIX on the DEC VAX and MVS, VM, and UNIX on the IBM mainframes. Tools specifically noted included implementations targeted to the the host machine, cross-reference listing generators, output listing formatters, interactive debuggers, and assembly language listing generators to show the relationships between the source code and the machine code generated by the compiler.

3.5.3 Configuration Management Tools

We evaluated each language in terms of what configuration management tools were available that could be used in conjunction with the language.

4.0 LANGUAGE COMPARISONS

4.1 ADA

4.1.1 Error Detection, Error Handling, and Error Containment

4.1.1.1 Design Translation

Ada provides a relatively small number of operations for expressing applications design. However, it does provide a variety of data types and modules, and it allows the user to define a variety of user data types and operations. At the same time, Ada syntax uses English-like, understandable primitives which assist the programmer in producing readable, understandable code.

The data types Ada provides for applications include integer, float, boolean, and character. Built-in operations on these data types are limited to basic arithmetic operations, logical operations, relational operations, and integer exponentiation. No built-in functions or operations are provided for other exponential or trigonometric operations.

User-defined types include not only array and Cartesian product aggregates, which may be nested, but also address data types for linked lists and dynamic types, range constraints, and enumerated lists. Integer precision is determined by the range of the integer type. For float data types, floating point and fixed point, precision is expressed by an explicit precision specification. The built-in data types are syntactically equivalent to user-defined data types and can be redefined by the programmer. However, though two data types may have exactly the same representation, i.e., both are defined as an integer type, they are considered to be distinct and may not be used interchangeably.

The language provides pre-defined operations for user-defined types. Concatenation is provided for array and string types. Operations for enumeration types and range types include membership tests and functions to compute the first, last, next, and previous value of a list or range.

User-defined operations or functions may be defined accepting any number of operators or arguments. Built-in operators, also, are syntactically equivalent to user-defined operations and may be re-defined. Any data type may be used as parameters or the returned value for a user-defined operation or function.

The basic module types, function, procedure, block, data block, and task, are provided by the Ada language. The data block can be used to encapsulate any and all definitions, including types, data, and other modules. A task module may be defined as user-defined type. All other named modules may not be used as data types. However, they may be used to define generic modules. Generic modules may be used to define modules in a manner analogous to the way data types define data objects. Modules defined from the same generic module differ only in the data types of designated

data objects within the modules. Groups of statements may be encapsulated by conditional control structures, by iterative control structures, and by simple block structures. The latter may be used to limit the scope of declarations and exception handling.

Ada allows identifier names to be as long as the length of one line; no name may extend over more than one line. Every character in the name is recognized by the language, including the underscore character which may act as a spacing character within a name. The language does not distinguish between upper and lower case characters in identifier names.

Comments, in Ada, end at the end of the line. They are delimited at the beginning of the comment by a double dash.

Ada uses sixty-six key words, of which only six are not English words or do not correspond to their English meaning. Punctuation is used as follows:

:=	assignment
;	statement terminator
,	separator within lists
()	designates groups of syntactical elements
=>	designates case alternatives and parameter assignments
:	designates the declaration of a data object
.	separates hierarchies of names within a full name
..	specifies a range of values

Ada control structures are fairly English-like and easy to understand. The CASE statement, one example, is shown below:

```
CASE x_variable IS
 WHEN 'A'    => a_count := a_count + 1 ;
 WHEN 'B'    => b_count := b_count + 1 ;
 WHEN 'C'    => c_count := c_count + 1 ;
 WHEN OTHERS => other_count := other_count + 1;
 END CASE;
```

4.1.1.2 Static Error Detection

Ada provides the ability to detect all the compile-time errors listed in the criteria. However, Ada also provides constructs which may be used to circumvent this checking. Since the compile-time error checking is not strictly enforced, some effort is required to ensure good coding practices are carefully followed.

Ada can be a strongly typed language. The value of an expression cannot be assigned to a variable if they are of different types. Also, an expression or variable cannot be passed as an argument to a parameter of a different type. Any type, including user-defined types, may be explicitly converted (if possible) to any other type.

All user-defined identifiers must be explicitly declared. Type declarations for float and fixed must explicitly specify the precision of the type. Range specifications for enumerated list, integer, float, and fixed types may be declared, but, range constraints may not always be testable at compile-time.

Ada provides several constructs by which strong typing may be bypassed including subtypes, derived types, overloading, and unchecked conversion.

Subtypes allow a programmer to differentiate between type checking that occurs during compile-time and type checking that occurs during run-time. A subtype is defined using a type and possibly a constraint to limit the possible values for the subtype. Two different subtypes are implicitly convertible if their ranges overlap and they are based on the same type. Errors in conversion are only detectable at run-time.

Derived data types, similarly to subtypes, are defined based on an existing type. Derived types are implicitly convertible to the base type only when used as arguments or returned values for subroutines.

Overloading is the practice of using the same name for more than one subroutine or operation. For example, the multiplication operation, designated by an asterisk, "*", is used to signify not only integer-integer multiplication and float-float multiplication, but also integer-float and float-integer multiplications which both return a result of the float type. This effectively nullifies type checking between integer and float data types for the multiplication operation. In this case, overloading is useful for expressing the concept of multiplying float and integer values. However, the language cannot protect against misuse of overloading to bypass type checking.

Finally, type checking can be avoided through the use of the built-in function, UNCHECKED_CONVERSION, which performs assignments at the bit-level without converting from one type to another. The language maintains some degree of control by requiring the function to be explicitly declared for each ordered pair of types to be assigned. The type of the input and output for the function must match the types of the corresponding input and output arguments.

4.1.1.3 Dynamic Error Detection and Exception Handling

Ada provides built-in run-time error testing for a number of predefined error conditions. When one of the error conditions is detected, execution immediately branches to the end of the inner-most block of code where it is processed by a user-defined exception handler or passed to the next outer-most block of code as an exception.

An exception is a pre-defined data type in Ada with one operation that signals the exception. This allows a programmer to declare user-defined exceptions and exception tests. The pre-defined exceptions for which Ada provides tests are:

constraint error attempting to access a non-existent data object or to assign a value outside the range of a data type,

numeric error dividing by zero or data overflow,

program error attempting to execute a non-existent or unallocated module,

storage error attempting to allocate a data object in non-existent storage, and

tasking error causing an error during inter-task communication.

Exception handlers in Ada are always specified at the end of a block or module of code. Handlers correspond either to one or more specific exceptions or to all exceptions that have not been specified. After an exception handler has executed, the block or module containing the exception handler is exited.

If a block does not contain an exception handler, the exception is signaled in the block which invoked the initial block. The exception is propagated through invoking blocks until it is processed or the outermost block is terminated.

Ada does provide the means to selectively by-pass run-time error testing. Specific tests may be turned off, such as array index checking, accessing an unallocated data block, out-of-range value assignments, the divide by zero condition, and others.

4.1.1.4 Error Containment

Ada provides some control structures which potentially can produce errors that are difficult or impossible to detect and locate. In some cases it provides alternatives to these dangerous constructs. For others, it provides the programmer with constructs to control and limit the probability of coding undetectable errors.

The GOTO statement in Ada is limited to branching either within the block of code that contains it or out of the block or nested blocks of code that contain it. It is not allowed to branch into a block of code, even if the branch first goes out of a similar block of code. It also may not branch into or out of a module. Blocks of code include BEGIN-END blocks, IF-THEN-ELSE blocks, CASE alternatives, and loop blocks. Statement labels to be used as targets are punctuated with the syntax:

```
<<label_name>>
```

Alternative control structures to the GOTO include the EXIT statement for branching to the end of a containing block within a module, the RETURN

statement for branching to the end of the innermost containing module, IF-THEN-ELSE and CASE conditional statements, and iterative statements, such as a conditional loop, an iterative loop, and an infinite loop which can be terminated only by an EXIT, RETURN, or external interrupt. Both the conditional and, of course, the infinite loops have the potential to loop infinitely.

Ada offers multiple levels of scoping based on the nesting of modules and BEGIN-END blocks. The outermost level of scoping is the scope of interfaces external to a compilation unit. These interfaces must be explicitly specified to be included in the scope of a compilation unit. Identifiers from enclosing scopes can be hidden by re-declaring the identifier within an inner scope. However, any identifier may be uniquely expressed as a qualified name (specifying the name of the module containing the declaration of the identifier and the name of the identifier) to avoid being hidden by a declaration within an inner scope.

Ada requires call-by-value or call-by-value-result for the parameter passing of simple data objects. Parameter passing for aggregate data objects is not restricted to any parameter passing technique, including call-by-reference, call-by-value, or call-by-value-result. Code which depends on any of these techniques may execute differently on different implementations of the language. Such code is "technically" erroneous, but the language does not check for such parameter passing dependencies.

Ada provides two constructs that can be used for aliasing variables: address data types and renaming declarations. Renaming declarations allow the programmer to declare alias names for previously declared identifiers. More than one address object may point to the same data object.

Address objects are restricted to point to exactly one data type. Discriminate unions are also restricted to represent one data type. Any attempt to vary the definition of the data object by re-assigning the discriminating component will result in an error. To vary the definition, the entire object must be re-assigned.

4.1.2 Modularity and Separate Compilation

4.1.2.1 Types of Modules Provided

Ada provides procedure, function, task, block, and data modules. Any type of module may be nested within any other type of module. Task and block modules must be nested within other modules. The data module may encapsulate module definitions as well as data definitions.

4.1.2.2 Interfaces Between Modules

Except for the statement block module, each module in Ada has a corresponding interface specification. An interface specification defines how the remainder of a program can interact with the module. Procedure and function interface specifications contain the name of the module, the names and declarations of all parameters that may be passed to the module, and the value returned by the module if it is a function.

A task interface specification contains the name of the task and declarations of any entries to the task. A task entry declaration has the same syntax as a procedure interface specification. An entry is an interface through which a program may rendezvous with a task and, optionally, pass arguments via the entry parameters.

A package specification consists of declarations for the data types, data objects, and modules by which the package communicates with the remainder of the program. Module declarations are the same as interface specifications. Task and package specifications must be explicitly coded. Procedure and function specifications are derived automatically from the corresponding module definitions when the definitions are nested in any module other than a package module. The language checks at compile-time that all interface specifications match the corresponding definitions of the module.

4.1.2.3 Separate Compilation and Interface Management

Ada allows separate compilation of procedure, function, and package modules both as external modules and as internal modules. Package interface specifications may be compiled separately from the package itself. This allows a program that references a package to be developed separately from the package, provided that the interface is not changed. The package must match its corresponding interface specification to be compiled successfully.

For external modules, such as functions, procedures, and data modules, interface specifications must be compiled before they are referenced by other modules. Once compiled, they may be included in the scope of another module by coding the specification names at the beginning of the referencing module.

Internal modules may also be specified for separate compilation. The modules in which they are nested must be compiled first to establish the scope in which the internal module is to be compiled. The scope of external modules may be limited explicitly to such separately compiled internal modules without including the external module in the scope of the outer nesting module.

4.1.3 Provision for Real-Time System Programming Constructs

4.1.3.1 Real-Time Tasking Constructs

Ada provides constructs for performing real-time operations for concurrent modules. Although the constructs do not exactly match the operations stated in the criteria, they may be easily adapted to execute these operations.

The language provides modules, known as tasks, which can execute concurrently in either a logical or a physical sense. The main or initial procedure can execute concurrently with the tasks. Task execution is initiated by either declaring or allocating the task as a data object. Declared tasks are executed as soon as the program is loaded and initial-

ized. Execution of dynamically allocated tasks is initiated at the moment of allocation.

Tasks may be terminated by completing execution, by executing a TERMINATE statement, or by the execution of an ABORT statement. The TERMINATE statement is allowed only within the task to be terminated. Execution of a TERMINATE statement may be controlled externally via rendezvous as explained below. The ABORT statement may be executed by any module in whose scope the task is known.

A task may be suspended in four ways: by a rendezvous, by the internal execution of a DELAY statement, by an external interrupt, or by a higher priority task becoming ready for execution. The latter cause is implementation-dependent. Execution suspended by an external interrupt is resumed after the interrupt is serviced. A task suspended by a rendezvous, a DELAY statement, or a higher priority task is placed on a queue to await its turn for execution. The manner in which this queue is organized is also implementation-dependent.

A rendezvous is a point at which the execution of two modules can be synchronized. It is initiated by a module making a call, similar to a procedure call, to an entry in a task module. The rendezvous is completed when the called task entry completes the execution of the acceptance of the call. More than one statement may be executed as part of the acceptance sequence. During this acceptance, the calling task is suspended.

The calling task may also be suspended while waiting for the called task to accept the call. Alternatively, the execution of the accepting operation by the task entry may cause that task to be suspended. Constructs may be used to control the length of time the calling task is suspended before the call is accepted or canceled. The suspension of the called task may also be controlled by specifying the maximum time the task may be suspended, by conditional expressions, or by alternative accept sequences (a task may contain more than one entry) and/or the TERMINATE statement.

Information may be exchanged between concurrent modules through parameter passing during entry calls or through global data objects. The mechanism for entry call parameter passing is equivalent to procedure call parameter passing. Since an entry call is a synchronization between the two tasks, there is no need to protect the data objects from simultaneous access by the two modules. The parameters used by the task entry are viable only during the execution of the acceptance.

Global data objects are not automatically protected from simultaneous access. The ability to provide this protection is implementation-dependent. When an implementation allows protection to be specified, the protection is limited to data objects that may be accessed in one uninterruptable operation.

4.1.3.2 Ability to Manipulate Bits and Bytes

Ada does not provide built-in data types or operations for manipulating bits or bytes. The user can define operations for manipulating data

types that represent physical bits and bytes. Constructs for defining such data types are known as representation specifications. These may be provided on an implementation-dependent basis. When provided, the constructs must adhere to the defined standard syntax. They include the specification of the amount of storage for a data type, the specification of bit and word representation of a data type, the specification of the addresses of data types and modules, and the ability to encode machine language instructions.

An implementation may provide constructs to control the amount of storage in which data types are stored. These may be used to control how many bits and words are used to represent the data type and/or the packing density of aggregates of the data type. A PACK construct may be provided to minimize the unused memory space between components of data aggregates. The details of how this construct operates are implementation-dependent.

The bit and byte representation of data objects may be specified for enumerated lists and Cartesian product data types. Constructs can specify what bit pattern should be used to distinguish one value from another in an enumerated list data type. The representation of the components of a Cartesian product data type can be explicitly specified in terms of the word and bit positions to which components are mapped. In addition, an implementation may provide a procedure for assigning the bit-pattern of one data object to another without performing any conversion or testing whether the bit-pattern is legal for the assigned data type. This procedure must be explicitly declared once for each different combination of data types to be assigned in this manner. Though no type checking is performed during the assignment, the types of the arguments passed to the procedure are checked by the language.

Constructs may be provided to specify the explicit memory locations of one or more data objects and/or modules. This allows the programmer to control any bit or group of bits in addressable memory. There are no checks, however, on the effect this may have on the reliability of code.

An implementation may provide a data module named MACHINE_CODE. When provided, this module must define Cartesian product aggregates to represent the machine instructions for the target machine. Ada restricts the use of these Cartesian product data types to user-defined procedures. The procedures may not contain any data type or data object declarations. They also may not contain any code other than the machine-instruction data aggregates.

4.1.3.3 Input/Output Capabilities

The provision of I/O subroutines and data types by Ada are implementation-dependent in the same manner as representation specifications (See "Ability to Manipulate Bits and Bytes" above). When provided, these routines must adhere to the standard language syntax. They include functions, procedures, and data types to create, open, close, and interrogate files, and to execute stream and direct access I/O.

There are two predefined low level I/O procedures: SEND_CONTROL and RECEIVE_CONTROL. Both procedures accept two arguments, one to specify the

device and the other to specify the data to be sent to or received from the device. The data types of the parameters are implementation-dependent as are the actual procedures.

4.1.3.4 Exception Handling Capabilities

The language provides the ability to specify a task entry as an interrupt handler. The manner in which the entry call is made is implementation-dependent.

The address which receives control due to a given interrupt may be explicitly attached to a task entry immediately following the entry declaration. The interrupt is handled as a high priority entry call. If it requires immediate attention, it is treated as a conditional entry call. If the interrupt can wait, it is treated as a timed entry call. Queued interrupts are treated as regular entry calls.

Interrupt initiated entry calls have a higher priority than regular entry calls. They may bypass scheduling to be executed immediately. Data associated with an interrupt may be passed as an input parameter to the task entry.

4.1.4 Stability and Portability

4.1.4.1 Stability of the Language Standard

Ada is defined by the Military Standard 1815A of January 22, 1983. This is also the approved American National Standards Institute, Inc., standard as of February 17, 1983. Although it is the practice of ANSI not to change an approved standard for five years, the Department of Defense could change its standard at anytime. The DoD has made it known that, barring some unforeseen problem in implementing the language, they do not intend to initiate any changes.

4.1.4.2 Enforcement of the Language Standard

Ada is a registered trademark of the U.S. Department of Defense through the Ada Joint Program Office (AJPO). The DoD has stated the policy that the term Ada may not be applied to any implementation unless it complies with MIL-STD-1815A. To enforce compliance, any and all implementations must pass a group of tests maintained and distributed by the Ada Validation Office (AVO) of AJPO. Each implementation must, thereafter, be re-validated within a year after validation and every year following. If no changes have been made to an implementation, then it may be allowed to delay re-validation for no more than two years. There are several suites of tests for each version of the test to help ensure that implementations strive to comply with the standard and not just the validation tests.

The validation tests do not currently cover the entire language standard. The current versions of the tests do not include representation specifications, interrupt handling, priority, MACHINE_CODE, and other constructs that tend to be implementation-dependent. Two more versions of the tests are currently under contract to be produced by the end of 1984.

The AVO has already refused validation to one implementation of the compiler due to failure to pass a current version of the tests.

Beyond validation, the AJPO has stated that it will consider legal action to disallow implementations that do not otherwise meet the language standard.

4.1.4.3 Demand for the Language

Under Secretary of Defense, Dr. R. J. DeLauer has distributed a memo [2] with the intention of making it a DoD directive that all DoD embedded computer systems that are designed after July 1, 1984, be required to use the Ada programming language. Two contracts have been granted by the Air Force and Army to produce Ada compilers and Ada programming support environments on the the DEC VAX and the IBM 370-series computers, respectively. Twenty to thirty other companies have begun work producing Ada compilers and Ada related tools for education and development, of which two companies and one University have succeeded in producing validated Ada compilers. The intense effort and investment by the DoD and private companies to provide Ada make it highly likely that it will be in high demand for the next several years.

4.1.5 Availability of Software Development Tools

There are only three validated implementations of Ada currently available. However, there are many major efforts to produce compilers and tools for the language. Several implementations potentially useful for AIPS are scheduled to be validated in 1984. Included with these implementations are Ada Programming Support Environment (APSE) of varying quality.

4.1.5.1 Availability of Implementations for AIPS Processors

There are currently no validated Ada compilers targeted to any of the candidate AIPS processors. Several implementations are under development, however, including:

Intel IAPx 186	produced by Intel for internal use, hosted on the target processor
Intel IAPx 286	produced by Intel for internal use, hosted on the target processor
Intel IAPx 432	produced by Intel for internal use, hosted on the target processor
Intel 8086	produced by SofTech, to be validated in 1984, hosted on the DEC VAX
MIL-STD-1750A	produced by Boeing, beginning development, to be validated in 1985,

Motorola MC68000	produced by Telesoft, to be validated in the spring of 1984, hosted on the target processor, others on the DEC VAX and IBM 370 mainframes to be validated in 1984
Motorola MC68010	produced by Telesoft, to be validated in 1984 or 1985, no plans to produce production quality compiler
National 16032	produced by National, beginning preliminary development,
National 32032	produced by National, beginning preliminary development,
Zilog Z8001	produced by Zilog, beginning preliminary development,
Zilog Z8002	produced by Zilog, beginning preliminary development,

4.1.5.2 Availability of General Tools

Several tool sets for software development of Ada programs are under development. Full information and schedules are not yet available for many that can be used for AIPS.

The Ada Language System (ALS) by SofTech is an APSE hosted on a DEC VAX consisting of a command language, Ada compiler targeted to the VAX, VAX assembler, VAX linker, VAX debugger, VAX assembly listing generator, data base management system, and text scripting tools for documentation, The command language is based on Ada. The data base management system is integrated with the Ada compiler. The tools in this system may be retargeted to to other processors and even used with other languages.

Other systems under development are similarly based on the ALS or on the UNIX operating system.

4.1.5.3 Configuration Management Tools

Configuration management is partially built into the language. However, tools will be necessary to control and maintain different versions of source code, object modules, load modules, test cases, and documentation.

The ALS provides configuration management for a large programming effort. A directory keeps track of different revisions of source code, object code, load modules, test data, and documentation text. Little used physical information can be off-loaded to secondary storage to relieve crowding on the development system.

4.2 C

4.2.1 Error Detection, Error Handling, and Error Containment

4.2.1.1 Design Translation

The language C provides some built-in and user-defined data types and offers a wide range of expression. However, enumerated lists and ranges are not provided as a type and user-defined data types cannot be passed or returned directly from a function. Also, though expressive, the language relies heavily on punctuation and operator symbols rather than key words. Though it is possible to write readable code, the language makes no attempt to enforce this practice.

The application data types provided by C include integer, float, boolean, and character. Operations are limited to basic arithmetic operations, the remainder operation, logical operations, and relational operations. Trigonometric and exponential functions are not provided by the language.

User-defined types include address data types and array and Cartesian product aggregates for all data types, including nested aggregates. Up to three relative levels of precision for float data types and three levels of ranges for integer data types may be specified. Precision and range specifications are optional and implementation-dependent. Parameters and returned values from functions, however, are limited to non-aggregate types (built-in types and address types). A user-defined type can be passed and returned only via an address type.

Modules provided include functions, procedures, block, and compilation units. Compilation units provide a means of encapsulating function, procedure, data, and type declarations. Groups of statements may be encapsulated in a block to be treated as a single statement and to limit the scoping of variables.

Identifier names distinguish between upper and lower case letters and the underscore character. No limit is placed on the length of a name, but only the first eight characters are significant in name identification.

Comments are delimited by the pair of double character symbols "/*" and "*/". There is no limit to the length of a comment.

The language has twenty-eight key words of which only six are not English. However, C relies heavily on symbolic punctuation. For example, blocks are delimited by braces. Other examples include the iterative FOR-loop:

```
FOR ( value = 1; value =12 ; value = value + 3 )
  some_procedure (value) ;
```

and the the switch expression:

```
value1 ? value2 : value3 ;
```

which returns the second or third value ("value2" or "value3" in the example above) depending on the boolean value of the first expression ("value1" in the example above). Yet another example is increment and decrement operations:

```
Y = Y + 1 ;
```

is equivalent to

```
Y += 1 ;
```

which is equivalent to

```
Y++ ; or ++Y ;
```

The same syntax may be used with the minus operator.

Some control structures do approach English in their appearance, such as WHILE loops:

```
WHILE (value=5)
  value = some_function ;
```

and

```
DO
read_something ;
WHILE (not_end_of_file)
```

and the IF statement:

```
IF (value < 5)
  value = 5 ;
 ELSE IF (value > 10)
  value = 10 ;
 ELSE
  value = next_value ;
```

The language provides a preprocessing facility for lexical substitutions. This allows a programmer to define and use understandable key words in place of required punctuation; the punctuation would be substituted for the key words prior to translation.

4.2.1.2 Static Error Detection

This language is weakly typed. Most built-in data types are convertible when mixed in an expression or an assignment. However, all data identifiers must be explicitly declared.

C allows character, integer, and float data types to be implicitly converted to integer or float, depending on whether or not a float data type is used in the expression or statement. Boolean types are not declared. In the correct context, integer types are interpreted as

boolean types. Float data types are not legal as subscripts. Each address data type is restricted to point to exactly one data type, both for assignment and parameter passing. However, type checking of parameters is only performed when the function or procedure definition and the invoking statement are in the same compilation unit.

Functions are implicitly assumed to return integer values unless explicitly declared otherwise.

4.2.1.3 Dynamic Error Detection and Exception Handling

C provides no language constructs for run-time error detection. Neither does it provide constructs for exception handling. Run-time errors are defined by the language, but the manner in which they are handled is implementation-dependent. In addition, some errors will produce undefined results.

4.2.1.4 Error Containment

The language C relies on data and control constructs that can produce errors that are difficult or impossible to detect or locate. Some primitive operations operate as side effects. The language does contain structured constructs to eliminate the use of the GOTO statement, but the power of address arithmetic is difficult to contain.

The GOTO statement in C is restricted to statement labels within the same scope as the GOTO statement. Statements are labeled by preceding them with a simple name followed by a colon. Structured control constructs include two different forms of conditional loops, an iterative loop, the IF-ELSE conditional, and the CASE conditional.

C provides multiple levels of scoping beginning with external variables as the outermost scope, and followed by compilation units, function definitions, and nestings of statement blocks. All functions are globally accessible. Global data objects defined in one compilation unit must be redeclared within the scope of another unit before they can be accessed by the second unit. Otherwise, there are no restrictions concerning access to variables declared in an outer scope except as follows: an identifier from an outer scope may be hidden by re-declaring the identifier within an inner scope.

Parameter passing is strictly call-by-value. This would eliminate the call-by-reference problems except that C relies on address data types to pass aggregate types to and from functions and procedures. Address types are restricted to point to their declared data types. However, address arithmetic allows the programmer to increment or decrement an address by the size of its data type. There are no restrictions on the number of increments and decrements. Also, there are no restrictions on the increments and decrements of array indices, which are effectively another form of address data types. This leads to an error containment problem as great as the unrestricted GOTO.

Address arithmetic, array indexing, integer, and float increment and decrement operations are available as side effects within expressions.

The operation may take place either before or after the expression is evaluated as specified by the programmer.

Discriminated unions are also easily abused in C. The programmer is free to change the discriminate to reinterpret a data object under a different data type definition.

4.2.2 Modularity and Separate Compilation

4.2.2.1 Types of Modules Provided

The language C provides three kinds of modules: function, block statement, and data modules. Function modules may be referenced as procedure modules by ignoring the value returned by the module.

The C data module is the compilation unit. A compilation unit contains data type, data object, and function declarations and definitions. A function module must be nested inside a compilation unit. Functions may not be nested inside functions, however.

Block statement modules must be nested inside functions. They may be nested within other block statement modules.

4.2.2.2 Interfaces Between Modules

The interfaces for functions are specified in the function definition. The values returned by a function are assumed to be integer unless the function is declared as another data type. Function interfaces are checked by the language only in the compilation unit containing the function definition. Between compilation units, there is no interface checking.

For the interface between compilation units, external data object and data type definitions must be explicitly declared to be included in the scope of another compilation unit. The names of external functions do not need to be declared unless they return a value other than integer. The language does no checking of the interfaces between compilation units.

Block statements do not require any interfaces since they are treated as a statement.

4.2.2.3 Separate Compilation and Interface Management

Only compilation units may be compiled separately. C does provide an include facility to copy source code from other files into a compilation unit. This can be used to ensure consistent declarations of global types and objects between different compilation units.

No facilities are provided for interface management since the language does no interface checking between compilation units.

4.2.3 Provision for Real-Time System Programming Constructs

The language does not provide very many constructs for system programming. However, the ability to address any part of memory via address operations and to easily manipulate bits allows any real-time functions or routines to be designed for an application, coded at the physical bit-level, and encapsulated within functions.

4.2.3.1 Real-time Tasking Constructs

C contains no real-time systems language constructs.

4.2.3.2 Ability to Manipulate Bits and Bytes

C provides a full array of basic bit operations including shift-left, shift-right, AND, OR, exclusive OR, and logical complement (not) to manipulate integer data types as bit-strings. The number of bits in each integer type is implementation-dependent.

The language allows access to any addressable memory through address arithmetic operations. Indices to arrays and address data types may be incremented and/or decremented without bounds to provide this access.

4.2.3.3 Input/Output Capabilities

C provides low level I/O subroutines READ and WRITE. Both accept two arguments: one to indicate the I/O device and the second for data.

The syntax and semantics for all other low level and higher level I/O functions are implementation-dependent. They include subroutines for opening and closing files and for seeking positions within files.

4.2.3.4 Exception Handling Capabilities

C provides no exception handling constructs.

4.2.4 Stability and Portability

4.2.4.1 Stability of the Language Standard

For the language C, we used as a standard the "C Reference Manual", the first appendix in the book The C Programming Language by Brian Kernighan and Dennis Ritchie, who are commonly considered the original designers of the language. Both people worked for Bell Laboratories, Inc., which also owns the licensing rights for UNIX, the operating system for which C was developed.

Almost all of the UNIX operating system is written in C. Implementations of the language C generally tend to be compatible to the extent that they can be used to implement UNIX. The language is also being used outside of UNIX environments, however, and there is no official standard which implementations can attempt to follow.

The American National Standards Institute (ANSI) has begun work on an ANSI standard for C. ANSI standards, though generally accepted, are applied voluntarily.

4.2.4.2 Enforcement of the Language Standard

There is currently no enforcement of any standard for the C language. Any enforcement of a future ANSI standard will be limited to implementations claiming to follow the ANSI standard.

4.2.4.3 Demand for the Language

The operating system UNIX, under licensing agreements with Bell Laboratories, is available on DEC computers, IBM mainframes, and a host of minicomputers and personal computers. The operating system and operating system tools are almost totally written in C. The current demand for the operating system assures that C will be available for many years on various systems.

4.2.5 Availability of Software Development Tools

C is generally available on many mainframes and microprocessors for implementing the UNIX operating system. Since the use of the language has been steadily growing over the last ten years, there is a mature set of tools available for developing and testing C code.

4.2.5.1 Availability of Implementations for AIPS Processors

The language is available on the following microprocessors:

- Intel IAPx 186
- Intel IAPx 286
- Motorola MC68000
- Motorola MC68010
- National 16032
- National 32032
- Zilog Z8001
- Zilog Z8002
- Zilog Z8003

4.2.5.2 Availability of General Tools

C is available on most DEC mainframe computers and on IBM 370 series computers. There are a number of tools that, when combined with an imple-

mentation of the language, create a more strongly typed language less prone to uncontrollable errors. These tools are written in C for the UNIX operating system and are portable between UNIX operating systems. They include an interactive C debugger, the program CREF that generates cross-reference listings for C source code, and the program LINT.

The program LINT searches C source code to enforce stronger typing and detect interfacing errors. The program detects and flags:

- implicit conversions,
- mismatches between arguments and parameters in function and procedure calls, even between separate compilation units,
- boolean expressions that have a constant value,
- unused variables, functions, and code,
- uninitialized variables, and
- non-portable language features.

The program also provides diagnostic messages which are generally superior to those commonly found in most implementations of C.

4.2.5.3 Configuration Management Tools

The C compiler and software tools are often purchased as a package with the UNIX operating system. UNIX provides a hierarchial filing system for software development and the MAKE program. This system is available on both DEC and IBM mainframes.

The MAKE program is used to keep track of the interdependencies of the modules used to create an entire program. When a newly modified module needs to be compiled, MAKE can automatically specify that all the modules dependent upon the modified module also be recompiled. This task is extremely useful for large software efforts containing many modules. (The MAKE program is not limited to the language C. It may be applied to other programming languages or mixtures of programming languages as well.)

4.3 FORTRAN 77

4.3.1 Error Detection, Error Handling, and Error Containment

4.3.1.1 Design Translation

The language FORTRAN 77 provides rudimentary data types for applications with a wide complement of operations for each data type. However, aside from arrays, there is no facility for defining new data types. Function, procedure, and data modules are provided, but no task modules exist in the language.

The applications oriented data types provided by FORTRAN 77 include integer, float, double precision float, complex, boolean, and character string. Bit manipulation functions allow integers to be used as bit-strings. Operations on these data types include basic arithmetic operations, integer and float exponentiation, logical operations, relational operations, and character string concatenation. In addition, FORTRAN 77 provides a wide range of trigonometric, exponential, hyperbolic, and other mathematical functions. It also provides character-string and bit-string manipulation functions.

The only user defined data types allowed by FORTRAN 77 are character string types and arrays of built-in data types. In both cases, the length of the character string type or array type may be explicitly specified. There is no provision for specifying the precision of other built-in data types. All data types may be passed as arguments to functions and subroutines and returned as values from a function with the exception that arrays may not be returned from a function.

FORTRAN 77 provides the programmer with program, function, and procedure modules, and data for encapsulating code and data. A special BLOCK DATA module is provided to initialize data in data modules. Groups of statements may be encapsulated by the IF-THEN-ELSE-END-IF block structure.

The length of identifiers in FORTRAN 77 is limited to six characters. Only letters and digits are allowed, and lower case is not distinguished from upper case.

Comments are terminated by the end of a line and denoted by either an asterisk, "*", or a "C" in the first character of a line.

There are over one hundred key words used in FORTRAN 77. Many of these are names of built-in functions, no longer than six characters, and are abbreviations for the functions they represent. Some are recognizable as engineering notation, such as SIN, COS, and LOG. However, others may appear cryptic depending on the background of the user, such as ERF, DBLE, AINT, and CONJG.

The semantics of control structures in FORTRAN 77 are not generally understandable from the English of their syntax. For example, conditional constructs such as:

```
    GO TO (100, 200, 315, 400) THIS1
100 iacnt = iacnt + 1
    GO TO 500
200 ibcnt = ibcnt + 1
    GO TO 500
315 iccnt = iccnt + 1
    GO TO 500
400 occnt = occnt + 1
500 CONTINUE
```

and

```
    IF (VALUE-6) 415, 512, 512
415 value = 5
512 CONTINUE
```

require knowledge of these structures to know that the numbers represent labels of statements to which control is to be switched. Neither do they convey the manner in which the language chooses the statement number in each list. The iterative loop structure is similarly cryptic.

```
DO 325 ILOOP = 1,12,3
```

Here, the variable ILOOP assumes the values 1, 4, 7, and 10 for each iteration of the statements following the DO statement up to the statement labeled 325.

4.3.1.2 Static Error Detection

FORTRAN 77 is a weakly typed language. There are almost no restrictions for implicit conversion between numeric data types. Integer, float, and complex data types are freely converted when mixed in expressions and assignments. Implicit type conversion is not allowed for parameter passing, for array subscripts, which must be integer, and between logical, character, and numeric data types. Implicit conversion during parameter passing cannot be checked at compile-time, however.

FORTRAN 77 also allows implicit identifier declaration. The programmer can use default conventions for implicit declaration, alter the implicit conventions, or specify that no implicit declarations are allowed.

4.3.1.3 Dynamic Error Detection and Exception Handling

FORTRAN 77 provides no constructs for detecting general run-time errors. If an error is detected during run-time, the language attempts to fix the error through a pre-defined standard actions. The programmer has the option of specifying alternate execution for errors that occur during the execution of specific I/O statements. This is implemented by specifying a statement label (within the I/O statement) to which execution should branch if an I/O error is detected.

4.3.1.4 Error Containment

FORTRAN 77 relies on control structures which can be extremely difficult to test. Its dependence on two levels of scoping, aliasing, and call-by-reference parameter passing create an environment with tremendous potential for side effects.

The GOTO statement, and forms of the GOTO statement, are the main control structures used in FORTRAN 77. The only restriction is that the target statement label be in the same scope, or module, as the GOTO statement. Statement labels are expressed as numbers in the first five character positions of a source code line. The only alternatives to the GOTO are the RETURN statement, the procedure CALL and function reference, and the block IF-THEN-ELSE-END-IF statement.

The language offers two levels of scoping: local and global. Global variables are grouped together in data modules. A subroutine may gain access to data in a data module by declaring the module internally. To be accessed, data within the module must be mapped to locally declared variables. The only restriction concerning what and how the data within a data module is accessed is that no data item be directly accessed as more than one data type. All procedures and functions are globally accessible without restriction.

Parameter passing in FORTRAN 77 is always call by reference, even for function modules.

The language provides an EQUIVALENCE statement to reference the same data object using different names and data types. This allows a data object to be referenced not only by two different names but also as two different data types.

4.3.2 Modularity and Separate Compilation

4.3.2.1 Types of Modules Provided

FORTRAN 77 provides procedure, function, data, data initialization, and program modules. The data module must be nested within the other modules. No other nesting is allowed.

The data initialization module defines the values to be used to initialize the data in a data module.

Each program must have one program module from which execution begins.

4.3.2.2 Interfaces Between Modules

The interfaces to procedure and function modules are derived from the definitions of the modules. The data module is itself an interface to any module that declares the data module. A program module has no data interface.

The language does not provide any checks between the interfaces of modules. There are no checks to insure that variable declarations and mappings to data modules are consistent between modules.

4.3.2.3 Separate Compilation and Interface Management

Program, function, and procedure modules may be compiled separately or together. No interface management is provided since no interface checking is performed.

4.3.3 Provision for Real-Time System Programming Constructs

4.3.3.1 Real-Time Tasking Constructs

The language FORTRAN 77 provides no real-time tasking constructs.

4.3.3.2 Ability to Manipulate Bits and Bytes

FORTRAN 77 allows integer data types to be manipulated as bit-strings through built-in functions.

There is no provision in the language to generate machine code instructions for manipulating special purpose registers on a target machine.

4.3.3.3 Input/Output Capabilities

FORTRAN 77 provides no low level I/O handling functions.

It does provide functions for direct access and stream I/O operations for the bit-representations of data objects. The programmer may open and close files for reading or writing.

4.3.3.4 Exception Handling Capabilities

FORTRAN 77 has no provisions for interrupt handling. The lack of an address type in the language makes it difficult to write constructs for handling interrupts.

4.3.4 Stability and Portability

4.3.4.1 Stability of the Language Standard

The standard we used for FORTRAN 77 is the ANSI standard X3.9-1978 known as FORTRAN 77. ANSI has made it a practice not to change their standards more than once every five years.

4.3.4.2 Enforcement of the Language Standard

The ANSI standard for FORTRAN 77 is applied voluntarily. ANSI may take legal action when an implementation makes false claims of following

the standard. Implementations may include disclaimers stating that they implement a subset of the standard or the standard with extensions.

4.3.4.3 Demand for the Language

FORTRAN has been in wide-spread use for many years on mainframe computers for engineering and other applications. New and revised implementations of the language are voluntarily providing all the features of FORTRAN 77. However, most implementations include extensions to the standard.

4.3.5 Availability of Software Development Tools

4.3.5.1 Availability of Implementations for AIPS Processors

We did not find implementations of FORTRAN 77 or FORTRAN 77 with extensions for any of the candidate microprocessors.

4.3.5.2 Availability of General Tools

Implementations that are extensions of FORTRAN 77 are available on the DEC VAX and IBM 370 mainframes. In both implementations, non-standard extensions can be flagged by the compiler. Both implementations include identifier cross-reference and and attribute listing generators.

4.3.5.3 Configuration Management Tools

We found no configuration management tools specifically made for FORTRAN 77. Rather, tools were tailored to specific implementations of the language or to specific target processors. Many implementations have an INCLUDE statement which may be used to copy text, such as declarations for data module variables, into compilation units. This feature can be used to guarantee data module agreement between different modules.

4.4 HAL/S

4.4.1 Error Detection, Error Handling, and Error Containment

4.4.1.1 Design Translation

The language HAL/S provides matrix and vector data types as well as integer, float, boolean, bit-string, and character-string. Operations for these types include basic arithmetic operations, integer and float exponentiation, logical operations for boolean and bit-strings, concatenation for character-strings and bit-strings, vector and matrix multiplication, and matrix exponentiation, transposition, and inversion. Built-in functions provide trigonometric, hyperbolic, exponential, and other mathematical operations for integer and float data types as well as character-string manipulation.

User defined types are limited to aggregates of the built-in types in the form of arrays and Cartesian products. A restricted address data type can be used to create linked lists and other structures. Strings, arrays, vectors, and matrices may be indexed by individual component, a slice of contiguous components, or, to produce an arrayed result, by an array of subscripts. User-defined functions and procedures can accept any data type as arguments, and functions can return any data type.

Procedures, functions, programs, and tasks are provided as module types. An update block module is used to protect groups of data from simultaneous access by more than one task at the same time. A COMPOOL module encapsulates data for interfacing purposes. Groups of statements can be encapsulated by a BEGIN-END block to be treated as one statement.

Identifier names are limited to thirty-two characters in length, including upper-case and lower-case letters, digits, and the underscore character. All thirty-two characters are significant for identifying a name.

Running comments are begun and terminated by the symbols "/*" and "*/", respectively, and may be limited in length by the implementation. An alternate form for comments may be designated by a "C" as the first character of a line and terminated by the end of the line.

HAL/S uses almost one hundred key words of which eighty-eight are English and have an English meaning. The language has fourteen operator symbols, some of which are used for subscripting arrays and strings. Alternatively, both subscripts and exponents can be specified using an optional three-line coding format. Each main line of code may be optionally specified by an "M" in the first character position of the line. An optional exponent line preceding a main line can be indicated by an "E" in the same position, and a subscript line immediately following a main line can be indicated by an "S". Exponents may appear in the exponent line immediately following the base of the exponent, and likewise for subscripts. For example:

```
E                  2    3
M  a_matrix     = x  + y  ;
S          2,3
```

The syntax for HAL/S declarations and control structures is relatively English-like. Declarations, for example, use English key words:

```
DECLARE ID_MATRIX MATRIX(3,3) INITIAL(1,0,0,0,1,0,0,0,1);
```

Conditional structures appear as follows:

```
IF value > 3 THEN
  value = 3 ;
 ELSE
  value = value + 1;
```

and

```
DO CASE test_number;
 ELSE other_count = other_count + 1;
  one_count = one_count + 1;          /* case 1 */
  two_count = two_count + 1;          /* case 2 */
  three_count = three_count + 1;      /* case 3 */
 END;
```

Iterative loops appear as follows:

```
DO WHILE (test_number > 3);
 test_number = some_function;
 END;
```

and

```
DO UNTIL (end_of_file) ;
  READ(5) character_data;
  END;
```

and

```
DO FOR x_number = 1 TO 12 BY 3;
  CALL some_procedure(input1, input2) ASSIGN(output1, output2);
  END;
```

The last example also demonstrates a procedure call. Arguments that are assigned values by the procedure are explicitly stated as such both by the call and by the procedure definition.

4.4.1.2 Static Error Detection

The HAL/S language is also a weakly typed language. The language does provide some control over the precision of data types, though no range control outside of precision, exists.

HAL/S allows implicit conversions between integer, float, character-string, and the primary components of vector and matrix data types. In fact, the primary components of vectors and matrices are considered equivalent types. Vectors and slices of matrices are also considered to be equivalent types when they contain the same number of components. User-defined data types, such as array and Cartesian product aggregates, are not implicitly convertible. Boolean and bit-string data types are also not implicitly convertible to other data types, though bit-strings of varying length are implicitly converted to the longer bit-string. These conversion rules apply to parameter passing as well as assignment and expression evaluation.

In addition, HAL/S provides a function which by-passes any type checking and conversion by allowing the programmer to extract and assign the bit-pattern representation of a data type.

HAL/S does provide the programmer with up to two levels of precision for integer, float, vector, and matrix data types. The lengths of array aggregates and strings may also be specified.

4.4.1.3 Dynamic Error Detection and Exception Handling

HAL/S provides constructs for run-time error detection and correction. However, the error conditions detected are implementation dependent. Those errors that are defined are classified in groups. Each error group is associated with a default system response, which may be to terminate the program, fix-up the error and continue, or ignore the error. Two statement constructs are provided for the programmer to specify alternatives to the standard system response.

The errors which an implementation may detect are each assigned a unique error code. Each group of errors is assigned a unique group code. Through the error handling construct, the programmer may specify a response to all errors, a specific group of errors, or one specific error per error handler. The error handler may specify that the error be ignored, that standard system action be taken, or it may specify code to be executed in place of standard system action. The error handler may be specified anywhere a statement may occur in the source code.

When an error occurs, execution is transferred according to the corresponding error handler that precedes the statement causing the error. If the error handler has turned off previous error handling or specifies system action, the system default for the error is executed and execution proceeds from the statement following the error. If the error handler specifies ignore, no action is taken and execution again proceeds from the statement following the error. If the handler specifies code to be executed, it is executed and execution proceeds from the statement following the error handler.

The programmer may also signal an error, whether system defined or user defined.

4.4.1.4 Error Containment

HAL/S provides structured alternatives to the GOTO statement. It offers multiple levels of scoping and the ability to restrict the scope of global variables. Both call-by-value and call-by-reference are used for parameter passing, but the latter is explicitly coded in both the procedure call statement and the procedure definition.

The HAL/S GOTO statement is limited to branches within the containing module. It may not branch into a block of statements, but it may branch within the containing block or out of one or more containing blocks. HAL/S also provides an EXIT statement to branch to the end of a block, the RETURN statement to branch to the end of a module, IF-THEN-ELSE and CASE statements, iterative loop statements, and conditional loop statements. The latter statements may potentially perform infinite loops.

The language provides multiple levels of scoping through nesting modules. External modules provide the outermost scope, but interfaces to these modules must be explicitly specified to include a compilation unit within this scope. Outer scope identifier names may be hidden by redeclaring the identifier within an inner scope.

HAL/S provides call-by-value parameter passing for input parameters. Call-by-reference is required for assign parameters. Both assign parameters and arguments must be explicitly specified by the ASSIGN key word in both the procedure definition and the procedure call.

Address data types may be used for aliasing, but each address type is restricted to exactly one object data type. The language preprocessor may also be used to alias identifier names.

4.4.2 Modularity and Separate Compilation

4.4.2.1 Types of Modules Provided

HAL/S provides the following kinds of modules: procedure, function, program, data, statement block, task, and update block. Task, update, and statement blocks must be nested within other modules. Procedure and function modules may be nested within program modules. Data and program modules may not be nested. Otherwise, program modules may be treated as task modules.

Update modules are used to access variables protected from simultaneous access by one or more concurrent tasks or programs. They are the only modules in which such variables may be modified or updated.

4.4.2.2 Interfaces Between Modules

HAL/S checks module interfaces at compile-time. All module invocations must be within the scope of the name of the module to be invoked. Procedure, function, and data modules are the only modules for which data interfaces need to be checked.

Data modules, known as COMPOOLs, are used as an interface for external data types and objects. All external data types and objects are declared in data modules.

4.4.2.3 Separate Compilation

Program, function, procedure, and data modules may be separately compiled as external modules in HAL/S. The language automatically generates an interface specification for each compilation. This interface specification must be specified at the beginning of any compilation unit that references the external module.

HAL/S automatically stores interface specifications in a library the first time a module is compiled and each time a compilation indicates that the interface has changed. When an interface is modified, all referencing modules must be recompiled. The specification may be manually generated to allow compilation of referencing modules before the referenced module has been compiled. However, the language does not provide any means to check the manually produced specification against the corresponding external module.

4.4.3 Provision for Real-Time System Programming Constructs

4.4.3.1 Real-Time Tasking Constructs

HAL/S provides task modules and constructs for controlling their execution. Inter-task communication is via global data objects. A special module exists to protect specified data objects from simultaneous access by more than one task module.

HAL/S provides both task and program modules as tasking modules. A task module, which must be nested within a program, is known only within the scope of the enclosing program. This limits the control of task modules to statements within the enclosing program and other tasks within the program.

Other than the first program which begins execution, all task and program execution is initiated by a SCHEDULE statement. The SCHEDULE statement can initiate execution immediately, at a particular time, after a period of time, or conditionally based on events. The statement can also cause cyclic initiation of execution based on time and specify the priority of a module to help decide conflicts in scheduling. The dependency of an initiated task module on the calling task module may also be specified.

A tasking module may be terminated by completing execution, by a CANCEL statement, or by a TERMINATE statement. The CANCEL statement may be executed by any other module to which the task module is known. The TERMINATE statement may only be executed by a module to which the tasking module is dependent. It also causes all dependent tasking modules under the terminated task module to be terminated.

The execution of a module can be suspended either by a higher priority module being scheduled for execution or by the WAIT statement. The WAIT

statement also specifies the conditions by which execution may be resumed, which may be a specified time, a period of time, an event, or the completion of execution by all dependent task modules.

Execution may be indirectly controlled by statements which change the priority of a module and by statements that alter events. Events are special purpose boolean data types which may be signaled (turned on and off) to trigger an event as well as act like boolean values.

All inter-task module communication is through global variables. The programmer may explicitly protect groups of variables from simultaneous access by more than one task module. Such variables are only accessible via update modules. Update modules may be nested inside procedures, functions, tasks, and programs.

4.4.3.2 Ability to Manipulate Bits and Bytes

HAL/S provides bit-string data types and operations for manipulating memory at the bit level. It does not provide constructs for explicitly specifying the addresses of data objects or modules. These constructs may be provided on an implementation-dependent basis.

Bit-level object manipulation is provided in HAL/S by the SUBBIT function and pseudo function. This function/pseudo function allows any integer, float, character, or bit data type or array of such data types to be referenced or assigned as a packed bit-string. Combined with bit-string data types and operations, this may be used to manipulate any simple data object or component of an aggregate data object.

The language does not provide constructs for explicitly specifying the address of bytes or words in memory. However, an implementation of the language may provide special operations via a construct (called %-macro) without altering the language or the language specification.

4.4.3.3 Input/Output Capabilities

HAL/S provides no low-level I/O capabilities. It does provide read and write constructs for direct access I/O. The language offers no direct control over creating, opening, or closing files.

4.4.3.4 Exception Handling Capabilities

HAL/S provides no constructs for connecting software modules to hardware interrupts.

4.4.4 Stability and Portability

4.4.4.1 Stability of the Language Standard

For the HAL/S language, we used the HAL/S language specification written by Intermetrics, Inc., for NASA Contract NAS9-13864 and modified under

a subcontract to IBM Corporation for NASA contract NAS 9-14444. The language and its specification belong to NASA and are subject to change according to its direction.

The language has been used almost exclusively to produce code for the space transportation system. It appears unlikely that the language standard will be changed in the near future.

4.4.4.2 Enforcement of the Language Standard

All existing implementations of HAL/S have been contracted directly or indirectly by NASA. NASA may enforce or modify the HAL/S standard to meet the requirements of stability and portability for the AIPS program.

4.4.4.3 Demand for the Language

HAL/S is the official language for NASA applications. It will remain in existence as long as NASA is willing to fund support for the language.

4.4.5 Availability of Software Development Tools

4.4.5.1 Availability of Implementations for AIPS Processors

There are currently no implementations of HAL/S targeted to any of the candidate AIPS processors. Intermetrics, Inc., has begun work on targeting HAL/S to a MIL-STD-1750A processor.

4.4.5.2 Availability of General Tools

HAL/S is available on the IBM 370 mainframe. Several tools have been constructed for managing software development with HAL/S for the Space Transportation System.

All implementations of the language include a cross-reference listing generator, an optional assembly language listing generator, and an intermediate code listing generator. A formatted listing is generated as part of the implementations which automatically indents and counts levels of nesting of nested modules and statement blocks, labels statements with statement numbers and line numbers from the source code, and expands the code into a three-line output format displaying exponents in an exponent line and subscripts in a subscript line where necessary. In addition, different data types, such as matrices, vectors, character, and bit types, are marked with identifying symbols in the exponent line while arrays and structures are surrounded by brackets and braces.

The implementations also provide a special linker which checks for interface consistency at link time and a load module statistics generator for generating a universal cross-reference listing for an entire load module. This linker also provides the capability of loading modules into explicit addresses in memory.

4.4.5.3 Configuration Management Tools

The language automatically generates, stores, and labels interface specifications in a library during compilation. This requires that modules must be compiled before modules that reference them. Other data management systems have been developed to store manual interfaces to allow the development of modules in any order. Some of these systems also store revisions of modules and allow the programmer to keep track of which modules need to be recompiled when an interface is modified.

4.5 JOVIAL J73

4.5.1 Error Detection, Error Handling, and Error Containment

4.5.1.1 Design Translation

The language JOVIAL J73 provides the programmer with some basic applications types and allows a wide range of user-defined types and operations. The language has a relatively large number of key words which are easily understandable but also relies on relatively cryptic punctuation and single character codes.

This language provides the basic applications types integer, float, boolean, bit-string, and character-string. Operations for these basic types include basic arithmetic operations, exponentiation for integer and float types, modulus for integer types, absolute value, logical operations, and relational operations. There are no trigonometric, exponential, or other mathematical operations or functions other than a function to indicate the sign of an arithmetic type.

User-defined types include array and Cartesian product aggregates, and enumerated lists. Varying precisions for integer, float, and fixed data types may also be specified. Address data types may be used to construct linked lists. In addition, a module type for encapsulating data is also provided. Built-in functions are provided to compute the next or previous elements in the in an enumerated list type. Built-in functions are also provided for address, array, and Cartesian product data types to indicate the size and bounds of data objects declared with these types. User-defined functions and procedures may accept or return any built-in or user-defined data type, including a module data type.

JOVIAL J73 provides program, function, procedure, and data modules. No task modules are provided. Data and type declarations and interfaces to procedure and function modules may be encapsulated within a data module for interface checking and access control. Groups of statements may be encapsulated within a BEGIN-END block to be treated as one statement.

Identifier names may contain from two to any number of characters including a dollar sign and apostrophe characters. Lower case characters are equivalent to upper case characters. Only the first thirty-two characters are used to distinguish one name from another. Iteration loop variable names are limited to one character in length. For external module names, implementations may recognize even fewer characters.

Comments are delimited at both ends by one of two characters, per cent signs (%) or double quotes ("). There is no length limit for a comment.

JOVIAL J73 utilizes eighty-seven key words, sixty-eight, of which, are English with English meanings. Built-in type identifiers, however, are all one character in length; a programmer may alias these with more descriptive identifier names. The syntax for control structures is divided roughly into half English key words and half punctuation. For example, the syntax for an iterative loop appears as follows:

```
FOR I : 1 BY 3 WHILE I < 12
  PROCEDURE'CALL'PASSING(I);
```

An example of a CASE statement is:

```
CASE x'variable;
 BEGIN
 (DEFAULT) other'count = other'count + 1;
 ('A')     a'count = a'count + 1;
 ('B')     b'count = b'count + 1;
 ('C')     c'count = c'count + 1;
 END
```

Semicolons are required to terminate simple statements, but not to terminate BEGIN-END blocks. Also, the user-defined values used in enumerated lists must have the syntax V(identifier) both when defined and when referenced in a statement.

4.5.1.2 Static Error Detection

JOVIAL J73 is a weakly typed language. Although it provides a wide range of precision specification for each data type, implicit conversions are allowed in most places where a compiler can perform the conversion.

Integer, fixed, float, and bit-string data types are implicitly convertible to float types. Otherwise, integer types are implicitly convertible to integer types, fixed types to fixed types, and bit-string types to bit-string types. Any address data type is implicitly convertible to an unrestricted address data type. Enumerated list types are implicitly convertible providing they contain exactly the same values. However, the ordering of the lists does not restrain the conversion.

Implicit conversion is disallowed for user-defined types, such as array and Cartesian product aggregates and data module types. However, array and Cartesian product aggregates may be explicitly converted to bit-string type and vice versa. Similarly, address data types may be explicitly converted to bit or integer data types and vice versa. Within user-defined aggregates, component declarations may overlay one another allowing the programmer to reference the same component object as two different types simultaneously.

JOVIAL J73 allows the programmer to specify explicitly how many bits of precision to use for integer, float, fixed, and bit-string types. In addition, the programmer may specify the manner in which values should be rounded, up or down, or truncated when a conversion to that data type occurs.

4.5.1.3 Dynamic Error Detection and Exception Handling

Run-time error conditions are defined in JOVIAL J73, but the manner in which they are handled is implementation dependent. Some error conditions

are specified as producing undefined results. There is a construct for handling user defined exceptions.

Procedure calls may optionally specify an abort clause. This clause specifies a statement label to which control is to be transferred if an ABORT statement is executed. The programmer may execute an ABORT statement within a procedure to generate a user exception. Execution branches to the statement specified by the first inner-most procedure call that specifies an abort option.

JOVIAL J73 also provides built-in functions and constants provided by each implementation which specify the maximum and minimum values of attributes for each data type in that implementation. These include maximum integer value, minimum integer value, maximum and minimum float value, maximum float precision, and similar constants for fixed data types as well. Built-in functions provide the same information for specific data types based on the type and precision of the argument passed to the function.

JOVIAL J73 has no built-in constructs for handling hardware exceptions or error conditions which may be defined for the language.

4.5.1.4 Error Containment

JOVIAL J73 provides a less restrictive form of the GOTO statement. It also provides explicit control of the scoping of global variables. Parameter passing may also be explicitly specified as call-by-value, call-by-value-result, and call-by-reference.

The GOTO statement in JOVIAL J73 is not allowed to branch outside of a containing module with one exception. Branching is allowed within a containing block of code or out of any number of containing blocks of code. It may not branch into a block of code even from within a block at the same or a higher level. The only time a GOTO may branch outside of a module is if the target statement is passed to the module as a parameter. In such a case, the exit from the module occurs without completing assignments to any assign parameters or returning any function value.

Alternatives to the GOTO statement include the EXIT statement to exit a block of code, the RETURN statement to exit a module, conditional statements such as IF-THEN-ELSE and CASE, iterative loop statements, and conditional loop statements. The latter have the potential to be infinite loops.

The language offers multiple levels of scoping dependent upon the nesting of modules. Global identifiers are defined in data modules. Access to each data module must be explicitly specified for any compilation unit and may explicitly restrict access to selected identifiers within the data module. Identifiers from enclosing scopes may also be hidden by the re-declaration of the identifiers within an inner scope.

JOVIAL J73 parameter passing defaults to call-by-value for simple input parameters, call-by-value-result for simple assign parameters, and

call-by-reference for aggregate parameters. However, the programmer may explicitly specify any of the above conventions for any parameter.

Aliasing in JOVIAL J73 may be achieved through textual substitution by the language preprocessor, through the use of address data types, and through the use of discriminated union. Address data types may be restricted or unrestricted. A data object may be referenced by more than one identifier and even as a different type through the use of unrestricted address data objects or discriminated union.

4.5.2 Modularity and Separate Compilation

4.5.2.1 Types of Modules Provided

JOVIAL J73 provides five basic modules: program, procedure, function, data, and statement block. Statement blocks must be nested within program, procedure, or function modules. Procedure and function modules may be nested within program, procedure, or function modules. Data modules must be compiled separately and may not be nested.

Each program contains exactly one program module where execution is to begin. Data modules, in JOVIAL J73, contain declarations for data types, data objects, and for function and procedure modules.

4.5.2.2 Interfaces Between Modules

The interface for a data module consists of all data objects declared as external, all external procedure and function interface declarations, and all data types. When the data module is specified as accessible to another module, the specification may limit which declarations are allowed within the scope of the referencing module.

External procedures and functions may be used by other modules only if their corresponding interfaces are declared within the scope of the modules. These interfaces may be declared within the invoking module or within data modules that are included in the scope of the invoking module.

4.5.2.3 Separate Compilation and Interface Management

Program, procedure, function, and data modules may be compiled separately. Procedure and function modules must be defined as external modules to be referenced outside the compilation unit. Data objects in data modules must also be defined as external objects to be referenced outside the module.

The language saves the interface of each data module compilation. To reference an identifier within the data module, a compilation unit must specify the name of the data module using an INCLUDE directive at the beginning of the compilation unit. By including a data module with a compilation unit containing external procedures and functions, a programmer can allow the language to check the procedure and function interface declarations within the data module against the corresponding procedure and function definitions in the compilation unit.

The external procedure and function module declarations can be checked against the corresponding module definitions only if the declarations are in a data module and the module is specified for reference in the corresponding compilation unit containing the procedure or function module definition. This is necessary only for interfaces to be automatically checked against definitions and is otherwise not required.

4.5.3 Provision for Real-Time System Programming Constructs

4.5.3.1 Real-Time Tasking Constructs

The language JOVIAL J73 provides no real-time tasking constructs.

4.5.3.2 Ability to Manipulate Bits and Bytes

JOVIAL J73 provides bit-string data types and operations including all logical operations and a substring facility to assign and reference slices of bit-strings.

JOVIAL J73 allows an implementation to provide built-in procedures and functions to implement target machine instructions which are otherwise not accessible via the language, such as instructions for loading and reading special control registers.

4.5.3.3 Input/Output Capabilities

JOVIAL J73 provides absolutely no I/O capability.

The language design requires I/O routines to be explicitly written for an application.

4.5.3.4 Exception Handling Capabilities

JOVIAL J73 has no interrupt handling constructs.

The language provides no simple means of connecting software modules to hardware interrupt addresses. This may be provided by implementation-dependent functions as described above in 4.5.3.2.

4.5.4 Stability and Portability

4.5.4.1 Stability of the Language Standard

The language JOVIAL J73 has been defined by the MIL-STD-1589B since June 6, 1980, and is required by the Air Force for all avionics embedded computer systems until Ada becomes available. Studies are being conducted by the Air Force concerning potential modifications to be made to the language, but no date has been set for the forthcoming release of MIL-STD-1589C. Extensions to the language are allowed in the form of implementation-dependent functions. These are generally limited to functions required for systems development rather than to extend the features of the language.

4.5.4.2 Enforcement of the Language Standard

Almost all JOVIAL J73 implementations are produced for the Air Force or for contractors working on Air Force projects. The Air Force requires all implementations used to produce code for these projects to have run a set of validation tests. These tests are maintained by the JOVIAL J73 Language Control Facility at Wright-Patterson Air Force Base which also publishes a list of implementations that are under development or are now available as validated compilers. Real extensions to the language, beyond implementation-dependent features, are limited to experiments, suggestions for MIL-STD-1589C, and/or implementations not used to produce code for the Air Force.

4.5.4.3 Demand for the Language

There are over twenty-eight validated JOVIAL J73 compilers under development or completed. These are hosted on DEC and IBM mainframe computers and targeted both to the host computer and to over seven different target processors including MIL-STD-1750 and 1750A, Intel 8086, Zilog Z8002, An/AYK-15, and Texas Instruments TI-990 and TI9900. The past investment of the Air Force in JOVIAL J73 software insures that the language will be available for a number of years.

4.5.5 Availability of Software Development Tools

4.5.5.1 Availability of Implementations for AIPS Processors

JOVIAL J73 is available on the following microprocessors through the listed companies:

Intel 8086 Produced by Proprietary Software Systems, this implementation is available as a cross-compiler hosted on an IBM 370 and on a DEC 10 mainframes.

MIL-STD-1750A Produced by Software Engineering Associates, this implementation is available as a cross-compiler hosted on an IBM 370 and on a DEC 10. Another implementation, produced by Proprietary Software Systems, is available as a cross-compiler hosted on an IBM 370 and on a DEC VAX. A third implementation, from the F16 Systems Programming Office (SPO) and hosted on the DEC VAX and on the IBM 370, is scheduled to be validated in 1984.

Zilog Z8001 Produced by Software Engineering Associates, this implementation is available as a cross-compiler hosted on a DEC 10 and on a DEC 20.

Zilog Z8002 Produced by Proprietary Software Systems this implementation, hosted on an IBM 370, is scheduled to be validated in 1984. Another implementation produced by SofTech is hosted on an IBM 370, on a DEC 10, and on a DEC 20. A third implementation is produced by Software Engineering Associates and is hosted on a DEC 10. A

fourth implementation, available from the F16 Systems Programming Office (SPO) and hosted on an IBM 370, is scheduled to be validated in 1984.

4.5.5.2 Availability of General Implementations

Tools for supporting JOVIAL J73 software development are generally contracted by the Air Force or developed by companies working on contracts for the Air Force or on JOVIAL J73 compilers.

Some implementations include cross-reference listing generators. Other tools, such as debuggers and assemblers, are developed for specific applications and/or target processors. A JOVIAL J73 interactive debugger is available from TRW that is hosted on a DEC 10.

4.5.5.3 Configuration Management Tools

A program support library for JOVIAL source code is available from SofTech on the IBM 370 mainframes.

4.6 PASCAL

4.6.1 Error Detection, Error Handling, and Error Containment

4.6.1.1 Design Translation

The language Pascal provides the data types integer, float, boolean, and character. Operations are limited to basic arithmetic operations, the trigonometric functions sine, cosine, and arc-tangent, the exponential functions exponentiation, natural log, and square root, and miscellaneous mathematical functions such as absolute value, square, decrement and increment for integers, and truncate and round for float.

User-defined data types include array, Cartesian product, and set aggregates, address types for dynamically allocated data types and linked lists, enumerated lists, and range specifications for integer data types. Built-in operations for enumerated lists and integer ranges include functions which return the "increment" or the "decrement" of the value passed. Enumerated list, boolean, and character data types also have a built-in function which returns the implementation integer value corresponding to the data type value passed. Set operations include union, intersection; tests for subset, superset, and membership; and set creation from a group of values. User-defined operations, functions and procedures, may accept arguments of any data type, including functions and procedures, and functions may return a value of any data type.

Pascal modules include procedures, functions, and a program module. No tasking or data modules are provided. Groups of statements may be encapsulated within a BEGIN-END block to be referenced as one statement.

Identifier names are limited to the length of a line and consist only of letters and digits; lower and upper case letters are equivalent. Only the first eight characters, however, are used to distinguish between names.

Comments are delimited by braces. There are no limits to the length of a comment.

Pascal uses over eighty key words of which over fifty are English. Though some operations have abbreviated names and punctuation, the syntax of control structures are generally English in appearance and meaning. For example, the conditionals:

```
IF value < 5 THEN
  value := 5
 ELSE IF value > 10 THEN
  value := 10
 ELSE
  value := next_value
```

and

```
CASE xvariable OF
  'A' : a_count := a_count + 1;
  'B' : b_count := b_count + 1;
  'C' : c_count := c_count + 1
  END
```

Iterative statements are particularly easy to understand:

```
WHILE value < 5 DO
  value = nextvalue
```

and

```
FOR index := 1 TO 12 DO
  thisprocedure
```

and

```
REPEAT
 thisprocedure;
 thatprocedure
  UNTIL value > 5
```

4.6.1.2 Static Error Detection

Pascal has been advertised as a strongly typed language. However, this assertion is not exactly true. Besides implicit conversion of integer types to float types, the language makes it difficult to disallow implicit conversions between similar user-defined types as explained below. The failure to distinguish between similar data types limits the ability to specify types that are conceptually unique though similar in implementation.

Float, character, boolean, and user-defined aggregate data types are not implicitly convertible. However, integer values are freely converted to float values when mixed with float values in an expression. Different user-defined data types are implicitly convertible, such as over-lapping integer ranges and enumeration data types that share the same user-defined values. Two different data types with the exact same definition are considered as one data type.

Pascal allows the programmer to constrain the range of values allowed for an integer type. However, the range constraint is not checkable at compile-time. The language does not provide the means to specify different sets of values for float types in terms of precision.

The use of variant Cartesian product aggregates cannot be checked at compile time. Such an aggregate may have a varying number of components or component types, depending on the value of a specified discriminating component. This construct allows a programmer to circumvent the type checking by reassigning the discriminating component before accessing the value of a variable.

4.6.1.3 Dynamic Error Detection and Exception Handling

PASCAL provides no constructs for detecting or handling run-time errors. The standard system action when a run-time error occurs is to abort the program.

4.6.1.4 Error Containment

Pascal has a GOTO statement, but provides a number of structured alternatives. Though it provides multiple levels of scoping, there are no restrictions on the use of a variable within its scope. Parameter passing is limited to call-by-value and call-by-reference, but these are explicitly specified only in the procedure definition.

The Pascal GOTO statement is restricted to branches within a module and within a block of code. It may also branch out of any number of containing code blocks, but is not allowed to branch into a code block even after branching out of a block at the same level. Alternatives to the GOTO include iterative loops, conditionals, such as IF-THEN-ELSE and CASE statements, and conditional loops. The conditional loops all have the potential to be infinite loops.

The language provides multiple levels of scoping dependent upon the nesting of modules. There are no restrictions on the use of identifiers within their respective scopes except that they may be hidden by redeclaration within a internally nested module.

All input parameters are call-by-value. All output parameters are call-by-reference. These are distinguished in the procedure definition but not in the procedure calls.

Aliasing may be achieved through address and discriminated union data types. Each address data type is restricted to point to exactly one data type. However, the discriminating component in a discriminated union type may be reassigned to change the definition of the type while referencing the same data object.

4.6.2 Modularity and Separate Compilation

4.6.2.1 Types of Modules Provided

Pascal provides program, function, and procedure modules. Each program is allowed exactly one program module. Function and procedure modules must be nested within the program module and can be nested within each other to any depth.

4.6.2.2 Interfaces Between Modules

The interfaces of the procedure and function modules are checked at compile time to be sure arguments and parameters match in both number and type.

4.6.2.3 Separate Compilation and Interface Management

Pascal has no provision for separate compilation.

4.6.3 Provision for Real-Time System Programming Constructs

4.6.3.1 Real-Time Tasking Constructs

The language Pascal provides no real-time tasking constructs.

4.6.3.2 Ability to Manipulate Bits and Bytes

Pascal provides boolean arrays and boolean operations. Boolean arrays may be packed such that access is provided to every bit in a physical word. There is no provision for explicitly specifying the location of a data object in memory.

There are no provisions in the language for specifying explicitly where a data object resides in memory. The language does not provide any means to execute implementation dependent machine instructions that might be needed for systems applications. There is no system level control of special control registers for a target implementation built into the language.

4.6.3.3 Input/Output Capabilities

Pascal has no low-level I/O capabilities.

All I/O in Pascal is limited to character stream I/O.

4.6.3.4 Exception Handling Capabilities

Pascal provides no means of connecting software modules to hardware interrupts.

4.6.4 Stability and Portability

4.6.4.1 Stability of the Language Standard

Two Pascal standards are currently recognized: the ANSI/IEEE 770X3.97-1982 and the standard published by the International Organization for Standardization (ISO). It is the established practice of ANSI not to change a standard more often than every five years.

For this study, we used the text by Grogono [3] which is based on the third draft of the British standard for Pascal. The British standard was the basis for the ISO standard.

4.6.4.2 Enforcement of the Language Standard

Both Pascal standards are applied voluntarily. Implementations of the language generally include extensions to the standards since the lack

of system constructs in Pascal make it very difficult to use otherwise.

Enforcement of the standard may occur only when an implementation claims to be standard or standard with extensions, and it does not meet the standard. When this fact is called to the attention of ANSI, the ANSI lawyers may decide to bring suit against the company making the claims. The need for ANSI to remain viable as a standards organization provides incentive for enforcement.

4.6.4.3 Demand for the Language

Pascal has quickly become the number one teaching language at American colleges and universities. The wide-spread popularity and familiarity of the language make it unlikely to fall into disuse in the near future.

4.6.5 Availability of Software Development Tools

4.6.5.1 Availability of Implementations for AIPS Processors

Implementations of Pascal where found for the following processors. Each implementation included extensions beyond the language standard.

- Intel IAPx 186
- Intel IAPx 286
- MIL-STD-1750A (MACDAC)
- Motorola MC68000
- Motorola MC68010
- National 16032
- National 32032

4.6.5.2 Availability of General Tools

Tools to support Pascal development are generally tailored for specific implementations of the language that contain extensions beyond the standard. We found no tools that were generally applicable to software development with standard Pascal.

4.6.5.3 Configuration Management Tools

We found no configuration management tools generally available for standard Pascal.

5.0 CONCLUSIONS

From a comparison of the six different languages, Ada was chosen as the language most suited for the AIPS program. The attributes of Ada would contribute more to the reliability, testability, and manageability of AIPS software development than any of the other languages. The widespread demand for Ada indicates that it will become available on a wide range of processors, as evidenced by the number of implementations currently under development. Together with the strict enforcement of the Ada standard by the Department of Defense, this implies that Ada will become the most portable of the languages in the next several years. All of the above result in increased development efficiency over both the short term and the long term.

The main drawbacks to choosing Ada are the availability of implementations of the language, the maturity of the Ada constructs for real time systems programming, allowable discrepancies between implementations, and the availability of software development tools. There are currently no validated implementations for any of the AIPS candidate processors, though several efforts are scheduled to be completed this year and other efforts have been or are being started. The validation tests will guarantee a relatively high level of reliability for any validated implementation. However, run-time efficiency, real-time operations, and optional constructs such as priority and the ability to use machine language will need to be evaluated for each individual implementation to determine how they will impact software design and implementation for the AIPS applications. New tools and development systems will also need to be evaluated as they become available.

5.1 ERROR DETECTION, ERROR HANDLING, AND ERROR CONTAINMENT

Ada is superior to any of the other languages in all areas of error detection and containment. The language also offers the widest facility for defining and encapsulating abstractions for data types and operations. Dangerous constructs that can cause errors which are difficult to detect can be isolated and exhaustively tested to minimize these errors.

Pascal and Ada are the most readable of the six languages. Both languages rely more on English key words and meanings than on symbols and punctuation. HAL, JOVIAL, C, and FORTRAN require a greater degree of memorization to understand the use and meanings of symbols, key words, and punctuation.

Ada provides a general facility to define and encapsulate data types and operations. None of the other languages make as clear a distinction between the interface to an abstract data type or operation and an implementation. Though the other languages offer more data types, such as vector and matrix in HAL/S, or operations, such as the built-in functions of FORTRAN 77, Ada allows the programmer not only to develop these data types

and operations, but to tailor them for a particular application or implementation without changing the interfaces.

Ada is also the most strongly typed language in the study. The level of explicitness for the definitions of abstractions is enforced by both compile-time and run-time testing of the boundaries of such definitions. HAL and JOVIAL were the only other languages to provide general run-time exception handling capabilities.

Each language provided control and data structures that could cause errors which are difficult to detect or contain. Ada allows some of these operations and structures, such as address data types and unchecked conversion, to be encapsulated within modules where they can be contained through run-time checks and testing. Other constructs, such as global variables, aggregate parameter passing, and renaming, may be required by some applications for efficiency of coding and execution, but must be designed, coded, and tested with extra caution to maintain reliability and flexibility between implementations.

5.2 MODULARITY AND SEPARATE COMPILATION

Ada is the only language that provided separate compilation for both internally nested modules and external modules. Uniquely, the language allows external modules to interface directly with internally nested modules that are compiled separately. As with HAL and JOVIAL J73, Ada allows access to global identifiers to be limited through interface specifications. Only Ada and JOVIAL J73 allow related data and module interfaces to be encapsulated as one interfacing unit.

5.3 PROVISION FOR REAL-TIME SYSTEM PROGRAMMING CONSTRUCTS

HAL/S and Ada are the only languages with built-in tasking constructs. The two languages present two different philosophies for real-time tasking control. Only HAL/S has been tested in the field. However, for either language, Ada or HAL/S, the behavior of tasking control constructs must be examined on an implementation by implementation basis to determine the applicability to a specific application.

HAL/S, JOVIAL J73, and Ada are the only languages with built-in constructs for accessing implementation dependent machine instructions. All these constructs are implementation dependent. Only Ada provided constructs for attaching interrupts to interrupt handlers at the source code level. C and Ada both provide low-level I/O constructs, which are also implementation dependent.

5.4 STABILITY AND PORTABILITY

Ada appears to be the most stable and, eventually, portable of the six languages due to high demand for the language and the DoD policy of strict standards enforcement. All implementations are required to undergo validation testing. A DoD wide office (AJPO) has been established to monitor the progress of language development and usage and to enforce the standard, even, if necessary, by legal means. The name of the language has been registered as a trademark of the U.S. Government to support this enforcement. JOVIAL J73 is the only other language for which implementations are required to undergo validation. The JOVIAL J73 standard is enforced by the Air Force. All current implementations of HAL/S are contracted and controlled by NASA. The other languages are defined or will be defined by ANSI standards, which are applied voluntarily. Enforcement is limited to implementations that claim to follow an ANSI standard.

Certain optional features in Ada detract from the stability of the language. These include the implementation of real time constructs and the ability to specify the machine representation of data types. However, these features are distinctly defined in the language standard and may be treated as non-portable features.

5.5 AVAILABILITY OF SOFTWARE DEVELOPMENT TOOLS

JOVIAL software development tools have generally been designed for specific sites and hardware or for specific applications. HAL software development tools have been implemented with the language, such as cross references, assembler listings, and global load module cross reference listing programs. Development tools for the language C are generally available and have been well tested and used. They are also generally portable between UNIX operating systems.

Ada software development tools have been produced and more are in development. However, due to their recent development, they have not been tested as thoroughly as general tools for C and for HAL. Two major efforts are underway to produce integrated software development environments for Ada: one on a DEC VAX and the other on an IBM 370-series mainframe. Each includes an implementation of the language, configuration management, documentation facilities, and facilities for building and attaching other tools, such as cross reference listing generators and assembly language listers.

BIBLIOGRAPHY

Ada, Reprints from Computer magazine, 1981, IEEE, 1981.

AIPS System Requirements, CSDL, AIPS Program, AIPS-83-50, August 30, 1983.

HAL/S Versus Ada, Tradeoff study for the space station program Intermetrics, Inc., Camino Center 11, 17625 El Camino Real, Suite 108, Houston, Texas 77058, (721) 480-4101, April 1, 1983.

VS Fortran Application Programming: Language Reference, IBM, Program Numbers 5748-F03, GC26-3986-3, Release 3.0, March, 1983.

Bardin, B.M.; Lane, D. S.; Huling, G., "Implementation of a Real Time Distributed Computer System in Ada", AIAA Paper No. 83-2407, 1983.

Booch, G., "Describing Software Design in Ada", Sigplan Notice., V 16, #9, ACM, September, 1981.

Brooks, F.P., The Mythical Man-Month, Addison-Wesley Publishing Company, ISBN 0-201-00650-2, 1975.

Bulman, David M., "Is Ada the Answer?", Pragmatics, Inc.

Carlson, W. E.; Druffel, Larry E.; Fisher, David A.; Whitaker, William A., "Introducing Ada", Proceeding of the 1980 Annual Conference, ACM, October 27-29, 1980.

Dacosta, Robert, "Babelism, Babbage's Booster, and Bernoulli's Numbers", Defense Science & Electronics, Rush Franklin Publishing, 201 East Campbell Avenue, Campbell, California, 95008.

Downes, V.A.; Goldsack, S.J., Programming Embedded Systems with Ada, Prentice-Hall International, ISBN 0-13-730010-7, 1982.

Ehrenfried, Lt. Daniel, "MIL-STD-1815A (ADA) and MIL-STD-1750A: Problems and Solutions", AFWAL/AAAF-2 WPAFB, OH 45433, (513) 255-2446, August, 1983.

Evans, A. Jr., "Comparison of Programming Languages: ADA, Praxis, Pascal, C.", Bolt, Beranek, & Newman, Inc., Cambridge, MA., UCRL-15346, 1981.

Felleman, P.G., AIPS System Requirements, CSDL, AIPS Program, AIPS-83-50, August 30, 1983.

Feuer, Alan R.; Gehani, Narain H., "A Comparison of the Programming Languages C and Pascal", Computing Surveys., Vol 14., No. 1, ACM, March, 1982.

Gehani, Narain, Ada -- An Advanced Introduction, Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, ISBN 0-13-003962-4, 1983.

Ghezzi, Carlo; Jazayeri, Mehdi, Programming Language Concepts, John Wiley & Sons, Inc., ISBN 0-471-08755-6, 1982.

Gilbert, Lucy, "Suitability of Ada Tasks for Distributed Computing", CSDL, September 1, 1983..

Grimsdale, R.L.; Halsall, F.; Shoja, G.C., "Some Experiences of Implementing the Ada Concurrency Facilities on a Distributed Multiprocessor Computer System", Software & Microsystems., V 1, #6, October, 1982.

Grogono, Peter, Programming in Pascal, Addison-Wesley Publishing Company, Inc., June 1, 1980.

Houghton, R.C., "A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE)",

Kernighan, Brian; Ritchie, Dennis, The C Programming Language, Prentice-Hall Inc., Englewood Cliffs, N.J. 07632, ISBN 0-13-110163-3, 1978.

Kernighan, Brian W., "Why Pascal is not my Favorite Programming Language", Bell Lab., Computing Science, Technical Report # 100, July 18, 1981.

Leach, Daniel M.; Satko, James E., "Implementation Languages for Data Abstractions", Phoenix Conference on Computers & Communications, (to be held) March 19, 1984.

Mueller, Frederick R.; Taft, Tucker S., "The Race Between C and Ada may have Two Winners", Electronic Design, Hayden Publishing Co., April 14, 1983.

Wulf, William A., "Trends in the Design and Implementation of Programming Language", Computer, IEEE Computer Society, Jan, 1980, Page 14-22.

Zuckerman, Susan Lana, "Problems with the Multitasking Facilities in the Ada Programming Language", Defense Comm. Eng. Center, Tech Note 16-81, May, 1981.

LIST OF REFERENCES

1. AIPS System Requirements, Charles Stark Draper Laboratory, Inc. AIPS-83-52 October 12, 1983..

2. DoD Directive 5000.31, Draft published in memo from Dr. Richard J. DeLauer, Under Secretary of Defense for Research and Engineering, June 10, 1983..

3. Programming in Pascal, Revised Edition Peter Grogono Addison-Wesley Publishing Company, Inc. Library of Congress Catalogue number QA76.73.p2g76 1980..